08 423402

PATENT APPLICATION

METHOD AND APPARATUS FOR SECURING EXECUTABLE PROGRAMS AGAINST COPYING

Inventors: Edward L. Schwartz
              1063 Morse Avenue, #10-104
              Sunnyvale, CA 94089
              Citizenship: U.S.A.

              Michael J. Gormish
              1070 Mercedes #4
              Los Altos, CA 94022
              Citizenship: U.S.A.

Assignees: RICOH COMPANY, LTD.
              3-6 Naka-Magome, 1-Chome
              Ota-Ku, Tokyo 143, Japan
              Incorporation: Japan

                     and

              RICOH CORPORATION
              2882 Sand Hill Road
              Menlo Park, CA 94025-7022, U.S.A.
              Incorporation: Delaware

Entity: Large

TOWNSEND and TOWNSEND KHOURIE and CREW
Steuart Street Tower, 20th Floor
One Market Plaza
San Francisco, California 94105
(415) 543-9600

5   METHOD AND APPARATUS FOR SECURING EXECUTABLE
PROGRAMS AGAINST COPYING

COPYRIGHT NOTICE

10   A portion of the disclosure of this patent document contains material which
is subject to copyright protection. The copyright owner has no objection to the
xerographic reproduction by anyone of the patent document or the patent disclosure in
exactly the form it appears in the Patent and Trademark Office patent file or records, but
otherwise reserves all copyright rights whatsoever.

15

BACKGROUND OF THE INVENTION

The present invention relates to the field of securing executable programs
against copying. More specifically, in one embodiment the invention provides security
against copying in an open hardware system where access to the processor executing the
20   program and the memory holding the program is assumed.

Securing computer programs (software) against unauthorized copying has
been a concern of software developers since software was sold as a separate product.
The difficulty lies in the fact that software is easily copied and any copy protection or
prevention scheme must allow for the eventual copying of portions of the software if it
25   is to be executed. Unless the software is secured in a chip (integrated circuit) which also
contains the microprocessor which will execute the software, the executed portions of the
software must pass from the distribution media to the processor along circuit lines which
are monitorable. Thus, for a program to be secure and still be useful to its intended
user, the program cannot be readily copyable in its generally available form or in the
30   form in which it is executed by the intended user.

Recently, with the increasing need for technical support from a program's
developer, the desire for complete documentation, and the fear of viruses, unauthorized
copying of some software, especially critical business software, has diminished.

However, where software needs no support or documentation and is used on systems where viruses cannot be transmitted, such as video game systems using video game cartridges with game software stored in read-only memory (ROM), unauthorized copying is still prevalent. All that is needed is an understanding of the circuitry used in the game

5   cartridge and a copy of the game program.

An additional concern of the makers of video games, who typically make video game consoles and wish to limit their use to games produced by licensed software producers, is not software copying, but video game console copying to produce consoles which will execute authorized game cartridges or unauthorized, but compatible, game

10   cartridges.

In an unprotected system, a copyist (i.e., a "software pirate" or other unauthorized analyzer or copier of the software) can easily copy program code if it is accessible. Program data, as used herein refers to the data necessary to run the program, which includes instructions (program code), tables of values and image data used to

15   generate screen images. Even if the program data is not easily accessible in its distributed form, a copyist might obtain it by observing a bus between the storage media which holds the program data and the processor to determine the program code. Thus, encryption of the program data alone does not provide real protection, since it must be decoded eventually to be used. Where the program data is stored on video game

20   cartridges and the processor is on a video game console, analyzing the program data is simplified, since the interface between the storage media and the processor is readily available without any hidden communication. In many video game consoles, the entire bus of the CPU is readily available for analysis. This particular problem, of course, extends to all forms of program storage media which are detachable, not just video game

25   cartridges.

Many copy protection systems are a deterrent to casual copyists, but not to determined copyists, who might be willing to spend large sums of money and time to break a copy protection scheme in order to be able to manufacture large numbers of unauthorized copies of a program. For some casual copyists, it is enough to include

30   software-only copy protection, such as the use of secret files or codes not normally accessed or observed by a casual copyist. Many casual copyists will also forgo copying when copying involves construction of cartridges, since this requires the ability to make plastic cases and circuit boards. However, the most determined copyists of cartridges

are those who plan to make large numbers of cartridges for sale and thus have the ability to make cartridges once the program data is copied.

Software-only copy protection systems, which might use an undocumented portion of the program data media to store hidden codes, generally rely on "security through obscurity" to prevent only those who are not aware of the copy methods from making workable copies. Therefore, when the goal is to stop large-scale and educated copyists, software-only protection is not viable. Fortunately, where the program data is distributed on media containing hardware elements, as is the case with video game cartridges, hardware copy protection can be included on the cartridge.

Many hardware protection systems rely on the presence of a hardware circuit or device which signals the existence of an authorized copy of the program. The program, when executed, runs a routine to check for the existence of the authorization device. If the authorization device is not present, the program refuses to continue or performs some other undesirable action. These protection systems are open to two methods of attack, both of which could render the protection ineffective.

In a first type of attack, a copyist would analyze the circuitry of the hardware authorization device to determine its essential elements and from that information make duplicate, unauthorized authorization devices. Even if the details of the authorization device are buried in a custom integrated circuit, the integrated circuit could be examined under a microscope layer-by-layer using a chemical peeling process to resolve the circuit features. The operation of the authorization device might also be observed by slowing down or speeding up both the authorization device circuitry and the processor to aid in the detailed analysis of one operation or the high-speed analysis of many passes over the program.

In a second type of attack, the copyist attempts to modify the software routines which check for the exists of the authorization device so that the routines always report back that the authorization device is in place, whether or not it actually is. With a readily-available logic analyzer attached to a microprocessor running a program, a copyist can run the processor at a slow speed and have the logic analyzer record all instructions executed by the microprocessor and all the data traffic to and from the microprocessor, then use this information to determine the flow of the program. If the flow of the program is recorded both with the authorization device in place (simulating an authorized use) and without the authorization device in place (simulating an

unauthorized use), the copyist can compare the flows and determine where in the program the decision is made as to whether the authorization device is in place. Once that location is determined, the software at that location could be modified so that the routine which tests for the presence of the authorization device never fails. This can

5   often be done by replacing one conditional jump instruction with an unconditional jump or a NOP (null operation).

        Therefore, what is needed is an apparatus which allows a processor to execute program code, over a possibly insecure bus, while requiring an impractical amount of work on the part of a copyist to reproduce the program data for use apart

10   from the apparatus or to reproduce the apparatus.

SUMMARY OF THE INVENTION

        The present invention provides a secure system for executing program code in an insecure environment while making it impractical to determine how to copy

15   the program code or associated data. In one embodiment of a secure system according to the present invention, a program memory contains encrypted program data (program instructions, data tables, digitized images, etc.) and security circuitry contained within an integrated circuit is provided for extracting the program data as it is needed by a processor. In various embodiments, the processor is a central processing unit (CPU), a

20   video pixel processor or other low-level CPU requiring program data. A portion of the processing steps which would be done by the processor in an insecure system is performed in this secure system within the secure circuitry using portions of the decrypted program data which are not provided to the processor. Program data is parsed based on a proper request to the security chip from the processor. The security chip

25   tracks which sections of the program memory are proper for the processor to be requesting based which program code in being executed. The security circuitry includes a key register in which a key value, needed to decrypt the program code, is stored. For security, a different key value can be used for each different program.

        Where a risk of chip peeling exists, the key might be stored in volatile

30   memory powered by a battery or stored as charge on capacitor, positioned and/or distributed on the security chip surface such that a chip peel breaks the source of power to the volatile memory well before the volatile memory can be reached.

In a specific embodiment, the security chip extracts the branch statements from the program instructions and stores them in an internal branch table after decryption and before providing the instructions to the processor. In a preferred embodiment, the branch statements are separated before being encrypted and stored in the program

5   memory. Because the possible flows of the program are known from the branch table, the branch table only need contain a listing of the branches which are imminent, thereby saving memory.

In various embodiments, the encryption is complex conventional encryption while in others, to save hardware, is simpler encryption such as XOR'ing

10 with the output of a pseudorandom number generator (PRNG). A number of additional security measures can be applied where needed. For example, if the security chip is positioned to read the processor bus, a tap of the processor bus can be provided so that the security chip can monitor all instruction fetches and data fetches from memory. For example, since the security chip provides all the branch information, the program flow

15 between branches is linear and deterministic. Thus, the security module could perform a checksum on all the bus activity between branches, compare it to a precompiled checksum and refuse to provide more branch information if the checksums do not match, as would be the case if the instructions provided to the processor had been modified in some way.

20   The security chip could also include a real-time clock, RC (resistor-capacitor) time constant circuit, or other dynamic logic circuit to confirm that the processor is executing instructions at an expected rate. This prevents a processor from being accelerated to speed up the process of running the program through all the possibilities needed to build an unauthorized branch table or from being slowed to

25 perform hardware analysis.

Furthermore, because the security chip maintains the branch table, it can calculate what the next branch is, so that the processor only need provide the values needed to evaluate whether to take a conditional branch. In order to handle return instructions, the security chip also maintains the program stack for the processor. This

30 security feature prevents the processor from requesting unexpected branch information.

To make a chosen text attack on the encrypted program data more difficult, the program data could be compressed first to remove patterns in the data.

The encryption and decryption could be done as conventional encryption/decryption. However, where low hardware cost is a priority, the encryptor could be just a data scrambler which rearranges the order of the bits or bytes of the program data according to the output of a PRNG. The data decryptor is then just a

5 series of buffers, a multiplexer and a demultiplexer. Where the security chip includes decompression, the buffers might already exist in the decompressor. If the scrambler is used in addition to other encryption, a chosen text attack is made more difficult, since the position of any word or bit in the data cannot be inferred. The PRNG is seeded by the key value or some agreed upon value dependent on the key. Because the file is

10 compressed, less robust encryption can be used. To further defend against analyses in which many different sections of the encrypted program data are compared with the corresponding decrypted data to determine the key value, a secondary key value which varies from section to section could be used. The secondary key value could be a value generated from data stored with the program data and the main key value. Alternatively,

15 a table of secondary keys could be stored with the program data or in the security chip, with the main key value used to select keys from the table.

In some applications, it is also desirable to prevent the operation of an authorized game cartridge on an unauthorized game console. For these applications, the game console is provided with a difficult to copy element and the security chip on the

20 game cartridge requires this element to be present before operating.

A further understanding of the nature and advantages of the inventions herein may be realized by reference to the remaining portions of the specification and the attached drawings.

25   BRIEF DESCRIPTION OF THE DRAWINGS

Figure 1 is a block diagram of a computer system according to the present invention wherein program data resides on a cartridge connected to a processor over a plug-in bus including a security chip on the cartridge;

Figure 2 is a more detailed block diagram of the security chip;

30 Figure 3 is a block diagram of a system used to encrypt programs onto encrypted memories which are used in the cartridges;

Figures 4A and 4B together show a flow chart of an execution of a program by the processor;

Figure 5 is a block diagram of a branch unit;

Figure 6 is a block diagram of a data stream scrambler; and

Figure 7 is a block diagram of a pseudorandom number generator.

DESCRIPTION OF THE PREFERRED EMBODIMENTS

Figure 1 shows a common application for the present invention, namely video games. It should be apparent after reading this disclosure that the invention is not limited to video games or programs stored on cartridges. In Figure 1, a game console 10 is shown with a game cartridge 12 about to be mated to game console 10 via a bus 14. We assume herein that a copyist is able to read the game program and game data as it is stored in game cartridge 12 and is also knows everything that goes on in game console 10. According to the preferred practice in the design of security devices, we also assume that the copyist knows all the details of any security circuits and algorithms except for any keys or passwords. Because the primary goal of the system is to allow a legitimate user to run the program, it is impossible to prevent the copyist from determining the outcome and flow of the program by running it with a given set of input data. Thus, the goal of the security of this system is to require that the program actually be run, at normal speed, and limit the amount of information about the program which can be inferred from one execution of the program with one input data set. The impracticality of determining the actual program code and data is based on the fact that the number of input data sets which would have to be run against the system is so high that the time needed to perform such an analysis is greater than, or comparable to, the time span in which the bulk of the legitimate sales of the program are made.

Game console 10 is shown with a processor 20, local memory 22, an output interface to a video display 16, an input interface from input devices 18, and a circuit clock 24. Game console 10 might include other elements not shown. The input devices 18 are shown generically, since they are not the subject of the present invention, but might include keyboards, joysticks, touch pads or sensor arrays. Video display 16 is typically a pixelated display (e.g., a raster monitor) which displays a two-dimensional array of pixels colored as indicated by processor 20 or an intermediate video memory (not shown). Local memory 22 stores variables used in the execution of the program as well as a current page of instructions of the program. Local memory 22 need not store the entire program at once, since the program can be paged into local memory 22 as

needed. As should be apparent from this description, the security system is described with reference to processor 20, the CPU of game system, however the present invention might also be used with a video processor, image processor or other processor present in the system.

5   Game cartridge 12 is shown with a security chip 30 and a ROM 32. Other forms of storage can be substituted for ROM 32, such as CD-ROM (compact disk ROM), diskette, flash memory, or even remote programs accessed via a network. ROM 32 contains the game program in an encrypted form. The game program so stored comprises program executable code, data tables, graphic images and other related objects

10   which are necessary or related to the operation of the game embodied by the cartridge. In a preferred embodiment, security chip 30 is a single integrated circuit with no secure data streams flowing on externally accessible pins.

          The present invention cannot protect against all attacks. For example, if a copyist obtains the game program in its unencrypted form from an unscrupulous

15   employee of the game maker, the present invention will not prevent the copyist from distributing the game in that form. It is assumed here that the unauthorized copyist only has access to a game console and a number of game cartridges. With this limited access, the copyist will attempt to analyze the operation of the console and cartridges using various analysis and control tools shown as tools 40 in Figure 1. An analysis tool

20   records signals, while a control tool changes the signals, often while an analysis tool is recording signals. Such tools include microprocessor trace analyzers, waveform generators, oscilloscopes, etc. Figure 1 shows various places within the game console 10 and game cartridge 12 the copyist could tap to analyze and control signals. Significantly, the copyist cannot tap onto internal lines of security chip 30 without chemically peeling

25   the security chip and working on a microscopic level.

          A tap 41 to circuit clock 24 might be used to slow down the processing speed of game console 10 to more easily analyze its operation, or used to speed up circuit clock 24 to more quickly execute many instructions to test different variations of the program. To render this tap ineffective, security chip 30 might also maintain a clock

30   whose speed is determined internal to security chip 30 and refuse to decrypt program data from ROM 32 if circuit clock 24 does not run at the correct speed.

          A tap 42 to ROM 32 provides the copyist with the contents of ROM 32, but these contents are not useful without the decryption provided by security chip 30. A

tap 44 on the data traffic on bus 14 or a tap 43 on the data traffic between processor 20 and local memory 22 might provide decrypted program information, but would only provide the instance of the program sequence which applies to one execution of the game instead of a sequence which is usable for different sets of input. Furthermore, that

5   information does not include branching instructions, as those are not sent to the processor, but are executed in security chip 30.

        The same is true for a tap 45 on the video display. While tap 45 might be able to record all the video signals, they would only correspond to one particular playing of the game. For example, if a game level containing unique graphics is never reached,

10   the graphics at that game level will never be sent to video display 16 and therefore cannot be obtained by tap 45.

        A determined copyist might use a tap 46 on the signals between input devices 18 and processor 20 to insert signals simulating different outcomes and events in the game. With enough different paths, a copyist might be able to determine all the

15   possibilities and work backwards to reconstruct the game program, but because the speed of the processor cannot be increased and due to the sheer number of possibilities, the copyist is not likely to be able to do this between the time a game is released to the public and the game sells enough copies to diffuse the copyist's potential market.

        Thus, even with all these taps, a copyist having access to only game

20   console 10 and game cartridge 12 cannot make a full, usable copy of the game program. A determined copyist might tap into the internals of security chip 30 itself, which would require careful removal of the casing of the security chip and a layer-by-layer analysis of security chip 30. However, even knowing the entire internal circuitry of security chip 30, the copyist will not obtain the volatile settings of the key needed for decryption of

25   ROM 32, since the memory holding the key is designed to lose power as layers are removed. If the memory is many layers below the layers required for power, the memory will be erased before its layer is available for analysis. In order to defeat security chip 30, the copyist must be able to analyze the contents of ROM 32 and the output of the security chip to derive the key. In addition to the key being different for

30   each different game (not necessarily each game cartridge containing that game), security chip 30 contains other features which make the cracking of the encryption without access to the internals of security chip 30 more difficult.

Figure 2 shows security chip 30 in more detail, including a bus unit 50 which couples security chip 30 to bus 14 and passes requests for program data to a translator 52. Translator 52 converts the address of the access request into an address location, or range of locations, in ROM 32. This address is sent over an address bus 54

5   to ROM 32, which returns the data on data bus 56. Alternatively, some locations of ROM 32 might be cached in cache 60, in which case translator 52 routes address information to cache 60, which in turn supplies the cached data.

Both sources of data are coupled to an input of a decryptor 62, which decrypts the data from ROM 32 using a key value supplied from a key register 64. As

10   explained below, different configurations for decryptor 62 are possible depending on the degree of security needed. Since the decryptor 62 is constrained to be, more or less, the inverse of the an encryptor used to encrypt the program data, the different configurations are discussed below in connection with the encryptor.

The output of decryptor 62 feeds to the input of a decompressor 68,

15   which is optional. Decompressor 68 in turn passes the data to a router 70. Router 70 includes an output for clear ROM data and an output for hidden data. The clear ROM data is unencrypted program code and data objects, but without branch information. The hidden data contains the branch information as well as other variables, such as checksums and expected execution times.

20   The hidden data output is coupled to an input of a computational unit 72, which handles the overall control of security chip 30. Computational unit 72 also includes a port for reading from, and writing to, private tables 74, an input for reading from a real-time clock 76, an input for receiving branch requests from bus unit 50, an output for branch responses coupled to bus unit 50, an input from bus unit 50 for bus

25   taps which provide information about activity occurring over a processor bus or bus 14, and a port for receiving and responding to queries from bus unit 50 about the propriety of requests for program data. The detailed operation of security chip 30 is described below in connection with Figure 4.

Figure 3 is a block diagram of a system 100 used to encrypt a game

30   program 112 onto an encrypted ROM 32 to be used in game cartridge 12. System 100 includes storage for game program 112 coupled to a branch separator 102 which outputs a secure program 114, a branch table 116, a file of checksum data 118 and a file of timing data 120. Storage is provided for these output data sets and that storage is

coupled to a compressor 104, which is optional but preferred. The output of compressor 104 is coupled to an input of an encryptor 108, which also has an input for receiving a key value from a key register 110. The output of encryptor 108 forms the contents of ROM 32.

5   As explained above, game program 112 comprises program executable code, data tables, graphic images and other related objects. A short example of program executable code, useful for explanation purposes only, is shown in Table 1. The program represented in Table 1 is a "clear" program in that all the information needed to run this program for any possible input is apparent from the program alone. Indeed, it

10   should be apparent from Table 1 that the program does nothing more than move the first ten entries of the array a[] into the corresponding locations in b[] for any possible input values (e.g., the values in a[]) .

Table 1. Clear Program	
Line #	Instruction
0	i=0
1	if (i > = 10) then goto 7
2	call 5
3	i = i + 1
4	goto 1
5	mov a[i], b[i]
6	return
7	return

The C program corresponding to the program of Table 1 is:

```
main {
        for (i=0; i<10; i++)
```

```
                move(i);
    }
    void move (int i) {
            b[i] = a[i];
5           }
```

If the program of Table 1 were supplied to branch separator 102, the secure program shown in Table 2 and the branch table shown in Table 3 would result.

Table 2. Secure Program		
Address	Line #	Instruction
0	0	i=0
1	1	mov i, br_req_arg[1]
2		br_req 0
3	5	mov a[i], b[i]
4	6	br_reg 1
5	2	br_req 2
6	7	br_req 3
7	3	i = i + 1
8	4	br_req 4

Table 3. Branch Table (Hidden Data)				
BR #	Type	Argument(s)	True Address	False Address
0	Cond'l	arg[1] > 10	6	5
1	Return	-	-	-
2	Call	-	3	-
3	Return	-	-	-
4	Goto	-	1	-

In generating the secure program, the order of the instructions could have been retained, so that the line numbers are in sequential order. However, if they were in order, the analysis of the addresses being passed to security chip 30 would indicate where jumps are being taken and not taken. For example, if the line numbers were in order in the secure program of Table 2, a request for jump address n, followed by a request for jump address n+1 would indicate that the jump associated with address n was not taken (otherwise a jump address other than n+1 would be the next jump address). To prevent this kind of analysis, the order of the lines in the secure program are scrambled. Since a true and false address is stored for each conditional jump, the code following a jump not taken does not need to sequentially follow the jump.

Appendices A, B and C are listings of longer examples of a clear program, its corresponding secure program and branch table, respectively. Appendix D is a listing of a program used in a software implementation of branch separator 102 (Figure 3). That program is written in the "awk" language which is commonly available on computers running the Unix operating system. As should be apparent, the program in Appendix B cannot be executed without the occasional reference to the branch table of Appendix C.

In some embodiments, branch separator 102 also calculates a checksum for the several instructions executed between each reference to the branch table. Since these several instructions do not contain any branches, they must be executed in the same order every time, and their checksum is easy to calculate. These checksums are stored as

5 checksum data 118. Similarly, in some embodiments, the execution time for the several instructions can be calculated and stored as timing data 120.

If more security against analysis is needed, instead of having the branches performed by security chip 30, security chip 30 could generate interrupts to implement taken branches. If this is done, a copyist would not detect branches not taken.

10 After secure program 114, branch table 116, checksum data 118 and timing data 120 are generated, if used, this information is compressed by compressor 104. In one embodiment, compressor 104 is the entropy coder shown in U.S. Patent No. 5,381,145, issued to Allen, Boliek, and Schwartz, and entitled "Method and Apparatus for Parallel Encoding and Decoding of Data." Compression is used not only

15 to allow more data to be stored in a fixed sized ROM, but is used to remove any patterns which might be present in the data, thereby making decryption without the key more difficult.

Encryptor 108 can take several forms. Where security is more of a priority than keeping the hardware cost low, encryptor 108 could be a Data Encryption

20 Standard (DES) encryptor, triple-DES, or a more secure encryption system as is known in the art of data security. Various embodiments of encryptor 108 can be used, depending on the laws of the country in which the game is sold and the country of its intended use, as well as a balancing of security needs and computational limitations. Where security of the encryption process is less of a priority than keeping hardware costs

25 down, several simple encryption circuits might be used. In one embodiment, encryption is merely the process of exclusive "OR"ing (XOR) the clear data with a stream of output bits from a pseudorandom number generator (PRNG). In another embodiment, the order of the clear data is reordered based on the output of the PRNG. With an incremental addition to the hardware cost, both of these methods could be used together.

30 The simple encryption is low-cost, since a PRNG can be easily build out of a few gates. See, for example, Figure 7 for a PRNG constructed from a shift register. Figure 6 shows a detailed diagram of the internal structure of for a data

scrambler 106. Data scrambler 106 uses a number of buffers, but the incremental cost of these buffers is zero where the buffers already exist as part of compressor 104.

In another low-cost variation, the encryption is combined with the compression. In this variation, the compression is entropy compression, which uses

5   tables of probability estimates to determine optimal codes to use. Since memory for these table is needed for compression anyway, using them for encryption adds no extra hardware. They are used for encryption by seeding them, initially or during the compression process, according to the key value or numbers based on the key value. An added benefit of this encryption scheme is that it prevents a known plaintext attack on the

10   encryption process since the compression process does not necessarily remove patterns in the data being compressed until the probability tables have had a chance to build up. With the key value providing the initialization for the probability tables, the compression process cannot as easily be analyzed.

After being encrypted and possibly compressed, the output of encryptor

15   108 is stored in the encrypted ROM 32. The operation of game console 10 and game cartridge 12 using encrypted ROM 32 will now be described, with reference to Figures 1-4. Figure 4 comprises Figure 4A and 4B, and together they show the steps taken by processor 20 and security chip 30 to execute a portion of the game program in a secure manner, beginning with a request for a page of instructions and/or data (Step S1).

20   For ease of implementation, the program data could be arranged in separately compressed data sets, with processor 20 requesting a page by specifying a pointer to a compressed data set, or just a selection of one page from a limited set of pages based on the last decoded page. For more security, each data set could be identified by a random ID associated with it at the time the program is encrypted.

25   The request for a page is passed over bus 14 to bus unit 50, which in turn queries computational unit 72 as to whether the request was proper (Step S2). Since computational unit 72 maintains the branch table, it can easily determine which instructions should and should not be requested by processor 20. This feature prevents a copyist from controlling the processor 20 such that it requests each and every block of

30   the program in a known order so that the copyist can assemble the entire program in the clear. If the request was not proper, computational unit 72 halts processor 20 (Step S3). Alternatively, computational unit 72 could cause some other effect, such as the erasure of the key value, erasure of ROM 32, cause the slow degradation of the data over time, or

other steps to frustrate further analysis. In some embodiments, computational unit 72 responds to a detected attack by altering the flow of the game such that if the copyist succeeds in deducing the flow of the game program, the deduced flow will be limited. For example, computational unit 72 could limit the program flow to just the first several

5 levels of the game.

If the request is proper, the page is extracted from ROM 32 (S4). To do this, the request is processed by bus unit 50 and sent to translator 52. Translator 52 is not necessary where the addresses used by the processor are the same the addresses used to access ROM 32. As Tables 1-2 indicate, the addresses do not always correspond.

10 The addresses also will need translation if the encryption alters the addresses for the data. Once the address of ROM 32 at which the requested page is stored is determined, that address is output on bus 54 to either ROM 32 or cache 60. In either case, the requested data is input to decryptor 62. The returned data might be program instructions or data objects. If the data is program instructions, corresponding entries of the branch

15 table are included with the data, along with the checksum and timing information, if used.

Decryptor 62 uses the key value from key register 64 to decrypt the data. Decryptor 62 is the inverse of encryptor 108, and the key value is either equal to, or the inverse of, the key value stored in key register 110, depending on the type of encryption

20 used. The decrypted data is then decompressed by decompressor 68. The effect of these elements is to reproduce sections of the data in data blocks 114, 116, 118 and 120 shown in Figure 3. This data is then separated into the secure program (clear ROM data) which is passed back to processor 20 via bus unit 50 and hidden data (branch table, checksums, timing data) which is passed to computational unit 72. Computational unit 72 stores this

25 data in private tables 74 (S5). Bus unit 50 passes the secure program page in the clear to processor 20 (S6). As explained above, the secure program page alone would not allow the copyist to duplicate the operation of the game program.

Once processor 20 has a page of program data, it executes the next instruction in that page (S7). Before executing the instruction, processor 20 checks the

30 instruction to see if it is a branch instruction (S8). If it is not a branch instruction, processor 20 executes the instruction (S9) and checks for further instructions in the page (S10). If more instructions are present, processor 20 gets the next instruction (looping

back to step S7), otherwise processor 20 requests the next page from ROM 32 (looping back to step S1).

On the other hand, if the instruction is a branch instruction, which processor 20 does not process, a branch request is passed to security chip 30 (S11). If the branch request is not proper, as determined in step S12, computational unit 72 halts processor 20 (S13). For a request to be proper, it must be expected, must occur at the time expected, and the bus checksum must be correct. The response from security chip 30 to a proper branch request is the address to which processor 20 should branch. Of course, for conditional branches, processor 20 will need to pass one or more arguments to security chip 30, which will calculate which of a true address and a false address to return (S14). Security chip 30 then passes the address back to processor 20 (S15), the processor jumps to that address (S16) and gets the instruction at that address (looping back to step S7). In this way, an address is provided for a branch without processor 20 ever being told what kind of branch it is or what all the possible branch addresses are.

Computational unit 72 is used both to process branch requests and to evaluate whether a request for a branch is proper. To do this, it uses a private branch table, which is stored as part of private tables 74. This branch table need not store all the branches, but just those which are upcoming. For each branch entry in the branch table, the following fields are maintained:

TYPE - the type of branch, selected from:

    1) unconditional jump

    2) conditional jump

    3) subroutine call

    4) subroutine return.

CONDITION - Only used with conditional jumps; indicates the condition tested.

ADDRESS 1 - For unconditional jumps and conditional jumps with a true condition, this address is the address to jump to; for calls, it is the called address; and for returns, it is not used.

ADDRESS 2 - Not used for unconditional jumps. For a conditional jump with a false condition, this is the address to jump to; for calls this is the return address, which is saved on a stack. This is not used for returns.

5   In some embodiments, processor 20 does not specify, and is never told, the index into the branch table for an upcoming branch. In those embodiments, the branch table tracks which branch index should follow the current branch. Of course, the next branch depends on which branch is taken, so that information is stored in the following two fields in those embodiments:

10

NEXT BRANCH 1 - Indicates the index of the branch in the program code which is the first branch in the code after ADDRESS 1.

NEXT BRANCH 2 - Indicates the index of the branch in the program
15   code which is the first branch in the code after ADDRESS 2.

CHECKSUM - The expected checksum for all the program code preceding the branch.

20   EXECUTION TIME - The expected execution time from the previous branch to the next branch.

PASSWORD - The password required to execute the current branch.

25   The TYPE field indicates the type of branch and, consequently, which other fields are used. If example, an entry for an unconditional branch (e.g., "goto 5") need not include a condition or a false condition address. Of course, in some systems other branches are possible, such as conditional calls and conditional returns. In some high-security systems, NOP branches might also be included in the secure program.

30   The CONDITION field might be expressed as an operand and a constant, for comparisons of a variable against the constant (e.g., "branch if (i $>=$ 10)") or just an operand for variable-to-variable comparisons (e.g., "branch if (x $<$ y)"). Where variables are needed for the comparison, they are passed by processor 20 to security chip

30 as part of the branch request. Processor 20 need not be informed of which condition is being applied, just how many and which variables to pass as arguments. In one embodiment, the TYPE field indicates which type of condition field is used and a VALUE field indicates the constant value where one is used.

5   The ADDRESS 1 and ADDRESS 2 fields supply the next address for the currently requested branch. For conditional branches, ADDRESS 1 is supplied if the condition is true, otherwise ADDRESS 2 is supplied. For unconditional branches, ADDRESS 2 is not used. For calls, ADDRESS 1 is the called address and is supplied to processor 20, while ADDRESS 2 is the address of the instruction following the call (i.e.,

10   the return address). That ADDRESS 2 value is placed on a stack for later use with the corresponding return branch. For a return, neither address field is used; the return address comes from the stack.

           Table 2 shows a secure program intended to be run by a processor such as processor 20. When the processor reaches a branch request (which replaces a branch in

15   the clear program), the processor makes a branch request and passes to the security chip the index of the branch request with the arguments necessary for the evaluation of a condition. For example, at address 4 in Table 2, a branch request is made for a specific branch index. The instruction "br_req 1" signals that entry 1 is the branch table is to be used. However, for some determined copyists, the index of the branch request might be

20   used to extract branch information. For example, by tracing enough "br_req 1" instructions, the copyist could determine that it is equivalent to a return instruction. In turn, each of the indexed branch requests can be analyzed to determine the type and condition for each branch.

           To make this sort of analysis more difficult, the indices in instructions can

25   be eliminated. Thus, instead of the instructions "br_req 1" and "br_req 2" being available to the processor, both of these are known only as "br_req". The indexing information is stored in the fields NEXT BRANCH 1 and NEXT BRANCH 2. Since all branches are controlled by computational unit 72, the branch request following the address of the current branch is known and thus is easily stored. The NEXT BRANCH

30   1 field contains the index of the next branch when ADDRESS 1 is the branch taken and the NEXT BRANCH 2 field contains the index for the next branch when ADDRESS 2 is the branch taken. For a call, ADDRESS 1 is the start of the called subroutine and ADDRESS 2 is the address of the instruction following the call instruction. Thus,

NEXT BRANCH 1 is the index of the first branch in the subroutine and NEXT BRANCH 2 is the index of the first branch following the ADDRESS 2 address. For calls, ADDRESS 2 and NEXT ADDRESS 2 are pushed onto a stack in computational unit 72.

5   Where used, the CHECKSUM, EXECUTION TIME and PASSWORD fields are used to determine whether the branch request is authorized. After a branch, bus information from bus taps is fed to computational unit 72, which checksums the bus data until a branch is found. The resulting checksum is compared to the stored CHECKSUM value. If these are different, computational unit 72 will take action to

10   prevent the further progress of the game. The checksum can be applied to all traffic over the processor bus except, of course, variable data.

        Similarly, real-time clock 76 is used to recorded the time between branches and that time is compared to the EXECUTION TIME value. The expected amount of time is easily determined if the processor clock rate is known, since there are

15   no intervening branches and the known number of instructions between the branches. Alternatively, a separate real-time clock is not needed. If a PRNG is used as part of the decryption process, it could be set to clock each instruction cycle, whether or not it is being used. That way, if extra instructions are inserted, the PRNG would lose sync with the data and corrupt it.

20   For particularly sensitive branches, a PASSWORD value could be assigned, where the processor must supply the password for the branch to be taken. The PASSWORD value might be calculated from a known combination of the state of the processor or memory contents of the local memory.

        Relying on the security chip for branch processing might take longer than

25   if the processor was executing a clear program. Where the execution of code is time-critical, the protection of selected sections of the program code could be disabled so that the processor does its own branch processing. Alternatively, portions of the processing necessary for the time-critical program code could be executed by the security chip to save processing time.

30   Figure 5 shows a branch unit 500 which is a part of computational unit 72 used to implement the above rules. Branch unit 500 receives an entry from branch table 502, data from a bus tap 504 and arguments from a DATA IN bus 506 and, based on those inputs, outputs either an error signal indicating an improper branch was requested

or a properly requested branch address. In some embodiments, the branch address is passed to processor 20, while in other embodiments the branch address is used to control which instructions are provided to processor 20 without ever informing the processor of the address for those instructions.

5   Branch unit operates as follows. Once an entry in branch table 502 is selected, the fields VALUE, TYPE, CHECKSUM, PASSWORD, TIMING, ADDRESS 1, ADDRESS 2, NEXT 1 and NEXT 2 are output. The VALUE field is the constant associated with a conditional jump with a comparison to a constant, and forms one input to subtractor 508. The other input to subtractor 508 comes from a register 510 which

10   holds the latest contents of the DATA IN bus. Subtractor 508 provides results for comparisons of a variable argument against a constant, and in some cases merely indicates whether the one input is greater than, equal to, or less than the other input. Subtractor 512 similarly compares the output of register 510 and the output of a second register 514 coupled to register 510. The outputs of both registers are provided to a

15   control section 516.

Control section 516 determines whether an error signal is to be output and also controls two multiplexers (muxes) 518, 520. The output of mux 518 is the branch address, selected from one of ADDRESS 1, ADDRESS 2 and a stack top value from a stack 522. The output of mux 520 is one of NEXT 1, NEXT 2, an initial value, and the

20   stack top value. The output of mux 520 indicates the index for the next branch and is fed back to an index input for branch table 502. The initial value is a pointer to the first branch in the chain of branches, so that the PENDING line is properly initialized.

Control section 516 determines which output of mux 518 is active based on its inputs, as indicated above: when the branch TYPE is an unconditional jump, a

25   call, or a conditional jump with a true condition, ADDRESS 1 is selected. ADDRESS 2 is selected for a conditional jump with a false condition and the stack top is selected for a return branch. In the case of a call, ADDRESS 2 is pushed onto the stack, for use with the next return.

Control section 516 also determines which output of mux 520: when the

30   branch TYPE is unconditional, a call, or a conditional jump with a true condition, NEXT 1 is selected and applied to the PENDING line. NEXT 2 is selected for a conditional jump with a false condition and the stack top is selected for a return branch.

Control section 516 outputs an error signal if the CHECKSUM value from branch table 502 does not match what checksum logic 524 calculates as the checksum from bus tap 504, if the execution time obtained by monitoring real-time clock 76 does not match the expected execution time indicated by the EXECUTION TIME (TIMING)

5   field, or if the password provided by processor 20 does not match the PASSWORD field.

Figure 6 shows scrambler 106 in more detail. Scrambler 106 comprises a demultiplexer (demux) 602, a mux 604, a pseudo-random number generator (PRNG) 606 and a number of buffers 608. Three buffers 608A, 608B and 608C are shown, however the number of buffers is not limited to three. The effect of scrambler 106 on an input

10   data stream is to rearrange the order of bits, bytes, words or blocks of the input data stream in a deterministic and reversible manner.

To do this, demux 602 parses the input data elements (bits, bytes, words or blocks) at its input to one of its outputs as determined by a current pseudorandom number. As the current pseudorandom number changes, the output to which the

15   elements are directed changes. Mux 604 combines the separated streams of elements into one stream as they are output by the buffers. The elements are reordered because they require different amounts of time to travel between demux 602 and mux 604, due to the arrangement of buffers 608. Each buffer 608 is either a first-in, first-out (FIFO) which alternates its head and tail, or is a FIFO which alternates as a last-in first-out

20   (LIFO) buffer. In the former case, each time an element is shifted into a buffer 608 from the left, an element is output to mux 604 from the right and each time an element is shifted into the buffer from the right, an element is output from the left. In the latter case, elements are pushed into the buffer from either end, but they are output from the right end. The end from which the element is shifted into the buffer 608 is controlled by

25   the values output by PRNG 606.

Thus, knowing the pseudorandom number sequence, one could discover the reordering pattern and reverse it. In order to discover the pseudorandom number sequence, the key value (stored in key register 64 of Figure 2 or key register 110 of Figure 3) must be known, since that key value acts as the seed for PRNG 606. Of

30   course, where hardware is at a premium, scrambler 106 could use portions of the decompressor, or the key value could be used to scramble or modify internal decompression tables, such as probability estimation or R-code tables.

Where hardware logic is especially at a premium, the pseudorandom number generator (PRNG) 700 shown in Figure 7 can be used. PRNG 700 requires only a key shift register 702, a maximal length sequence (MLS) shift register 704, two muxes 706, 708 and one XOR gate 710. Inputs to PRNG 700 are provided by a key clock 712,

5   an MLS clock 714, a KEY IN serial input, a KEY LOAD signal and an INIT/RUN signal. Once loaded, key shift register 702 can be used as key register 64 or 110, with the application of an uninterruptible power source 716.

Figure 7 shows the input to key shift register 702 being the output of mux 706, which is either the looped-back output of key shift register 702 or the KEY IN input

10   depending on whether the KEY LOAD signal at the select input of mux 706 is asserted or not. Figure 7 also shows the input to MLS shift register 704 being the output of mux 708 which is either the output of key shift register 702 or the output of XOR gate 710 depending on whether the INIT/RUN signal at the select input of mux 708 is set to INIT or RUN. The inputs to XOR gate 710 are the output of MLS shift register 704 and one

15   tap from a nonfinal stage of MLS shift register. The particular tap used and the number of stages in MLS shift register determine the length of the pseudorandom number sequence which results. For examples of sequence lengths and tap points, see Knuth, D.E., The Art of Computer Programming, 2d. Ed., 1981, pp. 27-29 and Table 1 therein. In one embodiment, the number of stages (flip-flops) for MLS shift register 704

20   is 98 and the number of bits in a key value is 98, however other lengths and taps work equally well. Of course, the number of bits in the key value should be large enough that a copyist cannot easily guess its value. A 98 stage MLS shift register will produce a sequence of bits which repeats only every $2^{98} - 1$ bits. With this many stages, the shift register need not be maximal length. One advantage to not having the shift register be

25   maximal length is that the set of taps for maximal length shift registers are known in the art and therefore a non-maximal length shift register would be more difficult to reverse engineer.

The key value is initially loaded into key shift register 702 by asserting the KEY LOAD signal, applying the key to the KEY IN input and clocking key clock 712

30   until the key value is loaded. Once the key value is loaded, the KEY LOAD signal is unasserted, so that the clocking of key clock 712 merely circulates the key value within key shift register 702. The KEY LOAD signal should be unasserted permanently, which could be done by having the key preceded by a leading "1" bit. That bit is then used to

set a flip-flop (not shown) when it reaches the output of key shift register 702. That flip-flop would control whether or not the KEY LOAD signal could be asserted and would also be powered by uninterruptible power source 716.

The key value is circulated in key shift register 702 to read it out. When
5   the INIT/RUN signal is set to INIT, the key value will be clocked into MLS shift register 704 by MLS clock 714, which clocks along with key clock 712. Once loaded, MLS shift register 704 will run and circulate its contents altered, of course, by XOR gate 710 to form a maximal length sequence of pseudorandom numbers, as is known in the art. The pseudorandom numbers of the sequence can be read out in parallel from MLS
10   shift register 704. With a low-cost PRNG such as the one just described, multiple keys become more feasible.

In summary, the above detailed description described a system for preventing the copying of program data by a copyist having access only to the end user portions of the hardware and/or software needed to run the program. While the
15   examples referred to a specific application of protecting game programs which are provided on game cartridges and used with game consoles, other applications were described. Furthermore, the invention is usable even where the program is provided primarily as software, so long as a small hardware component containing the security chip is provided. But one improvement of the present invention over the prior art is that
20   the program data is kept encrypted until decrypted by a security chip, and even then less than all of the program data is provided to a processor -- only the program code for executing the instance of the program whose flow is determined for the specific set of inputs provided by the user is made available. The data not provided to the processor is either provided only at a time known to be the appropriate time for the processor to be
25   requesting the program data, or is never provided to the processor. In the latter case, the security chip performs the operations which would have been performed by the processor had the security chip provided all the program data. In one embodiment, the information retained by the security chip is branching information. Thus, each time a processor encountered a branch instruction, that instruction could only be completed with
30   the assistance of the security chip.

The above description also described the internal structure and operation of the security chip, a system for generating encrypted program data, the interaction of the security chip and the processor during normal operation and operation while under attack

by a copyist, as well as a low-cost pseudorandom number generator based on a stored key value.

The above description is illustrative and not restrictive. Many variations of the invention will become apparent to those of skill in the art upon review of this disclosure. Merely by way of example, the above description described an embodiment of the invention which protected video game programs from unauthorized copying and use, however non-game applications of the invention follow from this disclosure. Other variations are discussed below.

In one particular embodiment of a video game according to the present invention, the security chip tracks the number of "lives" a player has left and the amount of time remaining until the player must move on to the next level. The security chip decrypts the program data one level at a time, and thus initially provides only the program data for the first level of the game. The program data contains initial values which are never shown in the clear outside the security chip and those values are used to set a timer for the level. As the player is playing the game, the program informs the security chip when certain events occur. If the information from the processor is as expected, the game is played normally. However, if the information is not as expected, time can be added to the timer or "lives" can be removed. If the time on the timer is extended, the player has more time to wait to get to the next level. If the time is extended often enough due to the program running differently than intended, the timer will never run down and the player will remain on the first level. One advantage to this approach over the security chip simply shutting down the processor is that it is not apparent when in the program the security chip first detects a problem.

The former approach is not without its advantages. If the processor is part of a fault-tolerant system, which might or might not be subject to attack by copyists, the security chip can be used as a means for preventing an errant processor from continuing once it has failed. The same security chip can be used to halt the processor or set off an alarm when improper execution is detected, but the assumption there is that the unexpected operation is caused by hardware or software failure instead of deliberate acts of a copyist.

As an example of non-game use, the processor might perform image processing. In image processing, a convolution might be required. A convolution is performed as a series of many multiply operations and an add operation. Since the

security chip can monitor the data bus of the processor, the security chip can perform the addition as the multiply results appear on the bus. When the sum is needed, the processor requests it from the security chip. If the processor is run without the security chip, the convolution will either be wrong or the processor will run slower since it will

5   have to do the accumulation itself.

Where the program data is compressed, elements of the decompressor might be used to contribute to the decryption process. For example, Figure 6 shows a data scrambler which could be formed from buffers of the decompressor. Also, where the decompressor is an entropy encoder, the security chip could use the key value for

10   form the initial parameters for the decompression process, such as the initial probability values. This has the additional advantage of preventing the one-to-one mapping of bits that usually occurs at the start of the entropy coder before enough bits have been received to infer probabilities from the bits themselves.

When the security chip detects a security violation it need not stop the

15   processor. Other options are to reset the processor after a random delay, output pseudorandom numbers as decoded data or adjust the PRNG by complement enabling or other means so that the PRNG slowly degrades the data being output.

The scope of the invention should, therefore, be determined not with reference to the above description, but instead should be determined with reference to the

20   appended claims along with their full scope of equivalents.

Appendix A. Example Clear (w/Branches) Assembly Language Program Listing

```
gcc2_compiled.:
___gnu_compiled_c:
.text
        .align 8
LC0:
        .ascii "%d\0"
        .align 8
LC1:
        .ascii " \0"
        .align 8
LC2:
        .ascii "\12\0"
        .align 4
        .global _print_it
        .proc 020
_print_it:
        !#PROLOGUE# 0
        save %sp,-112,%sp
        !#PROLOGUE# 1
        nop
        sethi %hi(_k),%o0
        ld [%o0+%lo(_k)],%o1
        add %o1,-1,%o0
        st %o0,[%fp-12]
L2:
        ld [%fp-12],%o0
        cmp %o0,0
        bge L5
        nop
        b L3
        nop
L5:
        ld [%fp-12],%o0
        mov %o0,%o2
        sll %o2,2,%o1
        sethi %hi(_x),%o0
        or %o0,%lo(_x),%o2
        sethi %hi(LC0),%o3
        or %o3,%lo(LC0),%o0
        ld [%o1+%o2],%o1
        call _printf,0
        nop
        ld [%fp-12],%o0
        and %o0,3,%o1
        cmp %o1,0
        bne L6
        nop
        sethi %hi(LC1),%o1
        or %o1,%lo(LC1),%o0
        call _printf,0
        nop
L6:
L4:
        ld [%fp-12],%o1
        add %o1,-1,%o0
        mov %o0,%o1
        st %o1,[%fp-12]
        b L2
        nop
L3:
        sethi %hi(LC2),%o1
        or %o1,%lo(LC2),%o0
```

```
              call _printf,0
              nop
      L1:
              ret
 5            restore
              .align 4
              .global _do_it
              .proc 020
      _do_it:
10            !#PROLOGUE# 0
              save %sp,-120,%sp
              !#PROLOGUE# 1
              st %i0,[%fp+68]
              sethi %hi(_x),%o0
15            mov 1,%o1
              st %o1,[%o0+%lo(_x)]
              mov 1,%o0
              st %o0,[%fp-12]
      L8:
20            sethi %hi(_k),%o0
              ld [%fp-12],%o1
              ld [%o0+%lo(_k)],%o0
              cmp %o1,%o0
              bl L11
25            nop
              b L9
              nop
      L11:
              sethi %hi(_x),%o0
30            mov 4,%o1
              or %o0,%lo(_x),%o2
              add %o1,%o2,%o0
              mov %o0,%o1
              st %g0,[%o1]
35    L10:
              ld [%fp-12],%o1
              add %o1,1,%o0
              mov %o0,%o1
              st %o1,[%fp-12]
40            b L8
              nop
      L9:
              call _print_it,0
              nop
45            st %g0,[%fp-16]
      L12:
              ld [%fp-16],%o0
              ld [%fp+68],%o1
              cmp %o0,%o1
50            bl L15
              nop
              b L13
              nop
      L15:
55            sethi %hi(_x),%o0
              ld [%o0+%lo(_x)],%o1
              st %o1,[%fp-20]
              sethi %hi(_k),%o0
              sethi %hi(_l),%o1
60            ld [%o0+%lo(_k)],%o0
              ld [%o1+%lo(_l)],%o1
              sub %o0,%o1,%o0
              mov %o0,%o1
              sll %o1,2,%o0
65            sethi %hi(_x),%o2
              or %o2,%lo(_x),%o1
              ld [%o0+%o1],%o0
              st %o0,[%fp-24]
```

```
            st %g0,[%fp-12]
    L16:
            sethi %hi(_k),%o0
            ld [%o0+%lo(_k)],%o1
 5          add %o1,-1,%o0
            ld [%fp-12],%o1
            cmp %o1,%o0
            bl L19
            nop
10          b L17
            nop
    L19:
            ld [%fp-12],%o0
            mov %o0,%o1
15          sll %o1,2,%o0
            sethi %hi(_x),%o2
            or %o2,%lo(_x),%o1
            ld [%fp-12],%o2
            mov %o2,%o3
20          sll %o3,2,%o2
            sethi %hi(_x+4),%o4
            or %o4,%lo(_x+4),%o3
            ld [%o2+%o3],%o2
            st %o2,[%o0+%o1]
25  L18:
            ld [%fp-12],%o1
            add %o1,1,%o0
            mov %o0,%o1
            st %o1,[%fp-12]
30          b L16
            nop
    L17:
            sethi %hi(_k),%o1
            ld [%o1+%lo(_k)],%o0
35          mov %o0,%o1
            sll %o1,2,%o0
            sethi %hi(_x-4),%o2
            or %o2,%lo(_x-4),%o1
            ld [%fp-20],%o2
40          ld [%fp-24],%o3
            xor %o2,%o3,%o4
            subcc %g0,%o4,%g0
            addx %g0,0,%o2
            st %o2,[%o0+%o1]
45          call _print_it,0
            nop
    L14:
            ld [%fp-16],%o1
            add %o1,1,%o0
50          mov %o0,%o1
            st %o1,[%fp-16]
            b L12
            nop
    L13:
55  L7:
            ret
            restore
            .align 8
    LC3:
60          .ascii "Usage: mlsg L K #lines\12\0"
            .align 8
    LC4:
            .ascii "\11"
            .ascii "0 < L < K < %d\12\0"
65          .align 4
            .global _usage
            .proc 020
    _usage:
```

```
          !#PROLOGUE# 0
          save %sp,-104,%sp
          !#PROLOGUE# 1
          sethi %hi(__iob+40),%o1
 5        or %o1,%lo(__iob+40),%o0
          sethi %hi(LC3),%o2
          or %o2,%lo(LC3),%o1
          call _fprintf,0
          nop
10        sethi %hi(__iob+40),%o1
          or %o1,%lo(__iob+40),%o0
          sethi %hi(LC4),%o2
          or %o2,%lo(LC4),%o1
          mov 100,%o2
15        call _fprintf,0 .
          nop
          mov 1,%o0
          call _exit,0
          nop
20  L20:
          ret
          restore
          .align 4
          .global _main
25        .proc 020
    _main:
          !#PROLOGUE# 0
          save %sp,-112,%sp
          !#PROLOGUE# 1
30        st %i0,[%fp+68]
          st %i1,[%fp+72]
          call ___main,0
          nop
          ld [%fp+68],%o0
35        cmp %o0,4
          be L22
          nop
          call _usage,0
          nop
40  L22:
          mov 4,%o0
          ld [%fp+72],%o1
          add %o0,%o1,%o0
          mov %o0,%o1
45        ld [%o1],%o0
          call _atoi,0
          nop
          sethi %hi(_l),%o1
          st %o0,[%o1+%lo(_l)]
50        mov 8,%o0
          ld [%fp+72],%o1
          add %o0,%o1,%o0
          mov %o0,%o1
          ld [%o1],%o0
55        call _atoi,0
          nop
          sethi %hi(_k),%o1
          st %o0,[%o1+%lo(_k)]
          mov 12,%o0
60        ld [%fp+72],%o1
          add %o0,%o1,%o0
          mov %o0,%o1
          ld [%o1],%o0
          call _atoi,0
65        nop
          st %o0,[%fp-12]
          sethi %hi(_l),%o0
          ld [%o0+%lo(_l)],%o1
```

```
        cmp %o1,0
        ble L24
        nop
        sethi %hi(_l),%o0
5       ld [%o0+%lo(_l)],%o1
        cmp %o1,98
        bg L24
        nop
        sethi %hi(_k),%o0
10      ld [%o0+%lo(_k)],%o1
        cmp %o1,1
        ble L24
        nop
        sethi %hi(_k),%o0
15      ld [%o0+%lo(_k)],%o1
        cmp %o1,99
        bg L24
        nop
        sethi %hi(_l),%o0
20      sethi %hi(_k),%o1
        ld [%o0+%lo(_l)],%o0
        ld [%o1+%lo(_k)],%o1
        cmp %o0,%o1
        bge L24
25      nop
        b L23
        nop
L24:
        call _usage,0
30      nop
L23:
        ld [%fp-12],%o0
        call _do_it,0
        nop
35 L21:
        ret
        restore
        .common _k,4,"bss"
        .common _l,4,"bss"
40      .common _x,400,"bss"

    ; End of Listing.
```

Appendix B. Corresponding Secure Assembly Language Program

```
     ------------------------------------------------------------
     Register %o7 is a register unused in the input assembly code
     ------------------------------------------------------------

     gcc2_compiled.:
     ___gnu_compiled_c:
     .text
             .align 8
LC0:
             .ascii "%d\0"
             .align 8
LC1:
             .ascii " \0"
             .align 8
LC2:
             .ascii "\12\0"
             .align 4
             .global _print_it
             .proc   020
_print_it:
             !#PROLOGUE# 0
             save %sp,-112,%sp
             !#PROLOGUE# 1
             nop
             sethi %hi(_k),%o0
             ld [%o0+%lo(_k)],%o1
             add %o1,-1,%o0
             st %o0,[%fp-12]
L2:
             ld [%fp-12],%o0
             sethi %hi(_SC_data),%o7
             st %o0,[%o7+%lo(_SC_data)]
             b _SC_branch+0
             nop
LSC0:
             b _SC_branch+1
             nop
L5:
             ld [%fp-12],%o0
             mov %o0,%o2
             sll %o2,2,%o1
             sethi %hi(_x),%o0
             or %o0,%lo(_x),%o2
```

```
            sethi %hi(LC0),%o3
            or %o3,%lo(LC0),%o0
            ld [%o1+%o2],%o1
            b _SC_branch+2
  5         nop
      LSC1:
            ld [%fp-12],%o0
            and %o0,3,%o1
            sethi %hi(_SC_data),%o7
 10         st %o1,[%o7+%lo(_SC_data)]
            b _SC_branch+3
            nop
      LSC2:
            sethi %hi(LC1),%o1
 15         or %o1,%lo(LC1),%o0
            b _SC_branch+4
            nop
      LSC3:
      L6:
 20   L4:
            ld [%fp-12],%o1
            add %o1,-1,%o0
            mov %o0,%o1
            st %o1,[%fp-12]
 25         b _SC_branch+5
            nop
      L3:
            sethi %hi(LC2),%o1
            or %o1,%lo(LC2),%o0
 30         b _SC_branch+6
            nop
      LSC4:
      L1:
            b _SC_branch+7
 35         restore
            .align 4
            .global _do_it
            .proc  020
      _do_it:
 40         !#PROLOGUE# 0
            save %sp,-120,%sp
            !#PROLOGUE# 1
            st %i0,[%fp+68]
            sethi %hi(_x),%o0
 45         mov 1,%o1
            st %o1,[%o0+%lo(_x)]
            mov 1,%o0
            st %o0,[%fp-12]
      L8:
```

```
        sethi %hi(_k),%o0
        ld [%fp-12],%o1
        ld [%o0+%lo(_k)],%o0
        sethi %hi(_SC_data),%o7
5       st %o1,[%o7+%lo(_SC_data)]
        st %o0,[%o7+%lo(_SC_data)]
        b _SC_branch+8
        nop
LSC5:
10      b _SC_branch+9
        nop
L11:
        sethi %hi(_x),%o0
        mov 4,%o1
15      or %o0,%lo(_x),%o2
        add %o1,%o2,%o0
        mov %o0,%o1
        st %g0,[%o1]
L10:
20      ld [%fp-12],%o1
        add %o1,1,%o0
        mov %o0,%o1
        st %o1,[%fp-12]
        b _SC_branch+10
25      nop
L9:
        b _SC_branch+11
        nop
LSC6:
30      st %g0,[%fp-16]
L12:
        ld [%fp-16],%o0
        ld [%fp+68],%o1
        sethi %hi(_SC_data),%o7
35      st %o0,[%o7+%lo(_SC_data)]
        st %o1,[%o7+%lo(_SC_data)]
        b _SC_branch+12
        nop
LSC7:
40      b _SC_branch+13
        nop
L15:
        sethi %hi(_x),%o0
        ld [%o0+%lo(_x)],%o1
45      st %o1,[%fp-20]
        sethi %hi(_k),%o0
        sethi %hi(_l),%o1
        ld [%o0+%lo(_k)],%o0
        ld [%o1+%lo(_l)],%o1
```

```
            sub  %o0,%o1,%o0
            mov  %o0,%o1
            sll  %o1,2,%o0
            sethi  %hi(_x),%o2
  5         or  %o2,%lo(_x),%o1
            ld  [%o0+%o1],%o0
            st  %o0,[%fp-24]
            st  %g0,[%fp-12]
      L16:
 10         sethi  %hi(_k),%o0
            ld  [%o0+%lo(_k)],%o1
            add  %o1,-1,%o0
            ld  [%fp-12],%o1
            sethi  %hi(_SC_data),%o7
 15         st  %o1,[%o7+%lo(_SC_data)]
            st  %o0,[%o7+%lo(_SC_data)]
            b  _SC_branch+14
            nop
      LSC8:
 20         b  _SC_branch+15
            nop
      L19:
            ld  [%fp-12],%o0
            mov  %o0,%o1
 25         sll  %o1,2,%o0
            sethi  %hi(_x),%o2
            or  %o2,%lo(_x),%o1
            ld  [%fp-12],%o2
            mov  %o2,%o3
 30         sll  %o3,2,%o2
            sethi  %hi(_x+4),%o4
            or  %o4,%lo(_x+4),%o3
            ld  [%o2+%o3],%o2
            st  %o2,[%o0+%o1]
 35   L18:
            ld  [%fp-12],%o1
            add  %o1,1,%o0
            mov  %o0,%o1
            st  %o1,[%fp-12]
 40         b  _SC_branch+16
            nop
      L17:
            sethi  %hi(_k),%o1
            ld  [%o1+%lo(_k)],%o0
 45         mov  %o0,%o1
            sll  %o1,2,%o0
            sethi  %hi(_x-4),%o2
            or  %o2,%lo(_x-4),%o1
            ld  [%fp-20],%o2
```

```
        ld [%fp-24],%o3
        xor %o2,%o3,%o4
        subcc %g0,%o4,%g0
        addx %g0,0,%o2
5       st %o2,[%o0+%o1]
        b _SC_branch+17
        nop
    LSC9:
    L14:
10      ld [%fp-16],%o1
        add %o1,1,%o0
        mov %o0,%o1
        st %o1,[%fp-16]
        b _SC_branch+18
15      nop
    L13:
    L7:
        b _SC_branch+19
        restore
20      .align 8
    LC3:
        .ascii "Usage: mlsg L K #lines\12\0"
        .align 8
    LC4:
25      .ascii "\11"
        .ascii "0 < L < K < %d\12\0"
        .align 4
        .global _usage
        .proc   020
30  _usage:
        !#PROLOGUE# 0
        save %sp,-104,%sp
        !#PROLOGUE# 1
        sethi %hi(__iob+40),%o1
35      or %o1,%lo(__iob+40),%o0
        sethi %hi(LC3),%o2
        or %o2,%lo(LC3),%o1
        b _SC_branch+20
        nop
40  LSC10:
        sethi %hi(__iob+40),%o1
        or %o1,%lo(__iob+40),%o0
        sethi %hi(LC4),%o2
        or %o2,%lo(LC4),%o1
45      mov 100,%o2
        b _SC_branch+21
        nop
    LSC11:
        mov 1,%o0
```

```
            b _SC_branch+22
            nop
     LSC12:
     L20:
5           b _SC_branch+23
            restore
            .align 4
            .global _main
            .proc   020
10   _main:
            !#PROLOGUE# 0
            save %sp,-112,%sp
            !#PROLOGUE# 1
            st %i0,[%fp+68]
15          st %i1,[%fp+72]
            b _SC_branch+24
            nop
     LSC13:
            ld [%fp+68],%o0
20          sethi %hi(_SC_data),%o7
            st %o0,[%o7+%lo(_SC_data)]
            b _SC_branch+25
            nop
     LSC14:
25          b _SC_branch+26
            nop
     LSC15:
     L22:
            mov 4,%o0
30          ld [%fp+72],%o1
            add %o0,%o1,%o0
            mov %o0,%o1
            ld [%o1],%o0
            b _SC_branch+27
35          nop
     LSC16:
            sethi %hi(_l),%o1
            st %o0,[%o1+%lo(_l)]
            mov 8,%o0
40          ld [%fp+72],%o1
            add %o0,%o1,%o0
            mov %o0,%o1
            ld [%o1],%o0
            b _SC_branch+28
45          nop
     LSC17:
            sethi %hi(_k),%o1
            st %o0,[%o1+%lo(_k)]
            mov 12,%o0
```

```
        ld [%fp+72],%o1
        add %o0,%o1,%o0
        mov %o0,%o1
        ld [%o1],%o0
5       b _SC_branch+29
        nop
    LSC18:
        st %o0,[%fp-12]
        sethi %hi(_l),%o0
10      ld [%o0+%lo(_l)],%o1
        sethi %hi(_SC_data),%o7
        st %o1,[%o7+%lo(_SC_data)]
        b _SC_branch+30
        nop
15  LSC19:
        sethi %hi(_l),%o0
        ld [%o0+%lo(_l)],%o1
        sethi %hi(_SC_data),%o7
        st %o1,[%o7+%lo(_SC_data)]
20      b _SC_branch+31
        nop
    LSC20:
        sethi %hi(_k),%o0
        ld [%o0+%lo(_k)],%o1
25      sethi %hi(_SC_data),%o7
        st %o1,[%o7+%lo(_SC_data)]
        b _SC_branch+32
        nop
    LSC21:
30      sethi %hi(_k),%o0
        ld [%o0+%lo(_k)],%o1
        sethi %hi(_SC_data),%o7
        st %o1,[%o7+%lo(_SC_data)]
        b _SC_branch+33
35      nop
    LSC22:
        sethi %hi(_l),%o0
        sethi %hi(_k),%o1
        ld [%o0+%lo(_l)],%o0
40      ld [%o1+%lo(_k)],%o1
        sethi %hi(_SC_data),%o7
        st %o0,[%o7+%lo(_SC_data)]
        st %o1,[%o7+%lo(_SC_data)]
        b _SC_branch+34
45      nop
    LSC23:
        b _SC_branch+35
        nop
    L24:
```

```
        b _SC_branch+36
        nop
LSC24:
L23:
        ld [%fp-12],%o0
        b _SC_branch+37
        nop
LSC25:
L21:
        b _SC_branch+38
        restore
        .common _k,4,"bss"
        .common _l,4,"bss"
        .common _x,400,"bss"
```

Appendix C. Corresponding Branch Table for Secure Assembly Program

5

```
      ----------------------------------------------

      Branch information in security chip

10    ----------------------------------------------

      ID      Label1      Label2    Cond.     Compare
      --      ------      ------    -----     -------
15    0       L5          LSC0:     bge       const 0
      1       L3                    always
      2       _printf,0   LSC1:     call
      3       L6          LSC2:     bne       const 0
      4       _printf,0   LSC3:     call
20    5       L2                    always
      6       _printf,0   LSC4:     call
      7                             return
      8       L11         LSC5:     bl        var
      9       L9                    always
25    10      L8                    always
      11      _print_it,0 LSC6:     call
      12      L15         LSC7:     bl        var
      13      L13                   always
      14      L19         LSC8:     bl        var
30    15      L17                   always
      16      L16                   always
      17      _print_it,0 LSC9:     call
      18      L12                   always
      19                            return
35    20      _fprintf,0  LSC10:    call
      21      _fprintf,0  LSC11:    call
      22      _exit,0     LSC12:    call
      23                            return
      24      _main,0     LSC13:    call
40    25      L22         LSC14:    be        const 4
      26      _usage,0    LSC15:    call
      27      _atoi,0     LSC16:    call
      28      _atoi,0     LSC17:    call
      29      _atoi,0     LSC18:    call
45    30      L24         LSC19:    ble       const 0
      31      L24         LSC20:    bg        const 98
      32      L24         LSC21:    ble       const 1
      33      L24         LSC22:    bg        const 99
      34      L24         LSC23:    bge       var
50    35      L23                   always
      36      _usage,0    LSC24:    call
      37      _do_it,0    LSC25:    call
      38                            return
```

Appendix D. Source Code Listing for Software Implementation of Branch Separator

```
 5    #!/bin/nawk -f $0 $*
      #
      # Awk program for automatically removing branch instructions from
      # SPARC assembly code generated by gcc so branches can be implemented
      # in security chip.
      #
10    # (C) 1994, 1995 RICOH Corporation.  All Rights Reserved.

      # ===================================================================

      function do_sc_label()
15    {
            sc_label = "LSC" new_label ":";
      }

      function do_branch1()
20    {
            if (opc_return[$1] != 1)
                  label[branch_count] = $2;
            if (condition[branch_count] == "") {
                  if (opc_return[$1] == 1)
25                      condition[branch_count] = "return";
                  else
                        condition[branch_count] = "always";
            }
            print "\tb  _SC_branch+" branch_count;
30          branch_count++;

      }

      # conditional branches and calls
35    # function do_branch2()
      {
            if (opc_call[$1] != 1) {

                  split(cmp_save,args,",");
40                print "\tsethi %hi(_SC_data),%o7"
                  print "\tst  " args[1] ",[%o7+%lo(_SC_data)]"
                  condition[branch_count] = $1
                  if (match (args[2],"^%")) {
                        print "\tst  " args[2] ",[%o7+%lo(_SC_data)]"
45                      compare[branch_count] = "var"
                  } else {
                        compare[branch_count] = "const " args[2];
                  }
            } else {
50                condition[branch_count] = "call";
            }
            label_flag = 1;
            label2[branch_count] = sc_label;

55          do_branch1();
      }

      # ===================================================================

60    # Change the initialization here to support parsing assembly code for
      # other processors.
      # function init_opcodes()
      {
            # Unconditional branches
65          opc_b1["b"] = 1;

            # Conditional branches
```

```
        opc_b2["bne"] = 1;       opc_b2["be"] = 1; opc_b2["bg"] = 1;
        opc_b2["ble"] = 1;       opc_b2["bge"] = 1;      opc_b2["bgu"] = 1;
        opc_b2["bleu"] = 1;      opc_b2["bcc"] = 1;      opc_b2["bcs"] = 1;
        opc_b2["bpos"] = 1;      opc_b2["bneg"] = 1;     opc_b2["bl"] = 1;

        # Calls
        opc_call["call"] = 1;

        # Returns
        opc_return["ret"] = 1;

        # Comparisions
        opc_compare["cmp"] = 1;

        # Instructions that can follow branch instructions
        opc_trail["nop"] = 1;
        opc_trail["restore"] = 1; # occurs after return
}

# ================================================================

BEGIN {
        print "Register %o7 must not be used in the input assembly code"
        print "------------------------------------------------------"
        print ""

        init_opcodes();

        cmp_flag = 0;
        label_flag = 0;

        branch_count = 0;
        new_label = 0;

        do_sc_label();
}

# ================================================================

END {
        print ""
        print "------------------------------------------------------"
        print ""
        print "Branch information in security chip"
        print ""
        print "------------------------------------------------------"
        print ""
        print "ID\tLabel1\t\tLabel2\tCond.\tCompare"
        print "--\t------\t\t------\t-----\t-------"
        print ""

        for (i=0;i<branch_count;i++) {
                l1 = label[i];
                while (length(l1) < 8)
                        l1 = l1 "        ";
                l2 = label2[i];
                if (l2 == "") l2 = " ";
                print i "\t" l1 "\t" l2 "\t" condition[i] "\t" compare[i]
        }
}

# ================================================================
#
# MAIN
#
# ================================================================

{
```

```
        if (cmp_flag) {
            cmp_flag = 0;
            if (opc_b2[$1] == 1) {
                do_branch2();
            } else {
                print "\tcmp " cmp_save;
            }
        } else if ((label_flag == 1) && (opc_trail[$1] == 1)) {
            print $0;
            print sc_label;
            label_flag = 0;
            new_label++;
            do_sc_label();
        }else if (opc_compare[$1] == 1) {
            cmp_flag = 1;
            cmp_save = $2;
        } else if (opc_b1[$1] == 1) {
            do_branch1();
        } else if (opc_call[$1] == 1) {
            do_branch2();
        } else if (opc_return[$1] == 1) {
            do_branch1();
        } else
            print $0;
    }
    /* End of Listing. */
```

Appendix E. Source Code Listing for Program to Test PRNG

```
5
     /******************************************************************
     *
     * mlsg.c
     * An example C program: Test maximal length sequence generator.
     *
10   * (C) 1994, 1995 RICOH Corporation.  All Rights Reserved.
     *
     ******************************************************************/

     #include <stdio.h>
     #define MAX 100
15
     int k, l;
     int x[MAX];   /* K Flip-flops are used in hardware */

     void print_it() {
20        int i;
          for (i=k-1;i>=0;i--) {
               printf("%d",x[i]);
               if (i%4 == 0) printf(" ");
          }
25        printf("\n");
     }

     void do_it(int n) {
          int i, j;
30        int t1, t2;
          x[0] = 1;
          for (i=1;i<k;i++)
               x[l] = 0;

35        print_it();

          for (j=0;j<n;j++) {
               t1 = x[0];
               /* t2 = x[1];  Alternative*/
40             t2 = x[k-1];
               for (i=0;i<k-1;i++)
                    x[i] = x[i+1];
               x[k-1] = (t1 != t2) ? 1 : 0; /* XOR gate is used in H/W */
               print_it();
45        }
     }

     void usage() {
          fprintf(stderr,"Usage: mlsg L K #lines\n");
50        fprintf(stderr,"\t0 < L < K < %d\n", MAX);
          exit(1);
     }

     void main (int argc, char **argv) {
55        int lines;

          if (argc != 4) usage();
          l = atoi(argv[1]);
          k = atoi(argv[2]);
60        lines = atoi(argv[3]);

          if ((l<1) || (l > MAX-2) || (k < 2) || (k > MAX-1) || (l >= k))
               usage();
          do_it(lines);
65   }
     /* End of Listing. */
```